(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 375 221 A1**

(12) EUROPEAN PATENT APPLICATION

(21) Application number: 89312818.1

(22) Date of filing: 08.12.89

(51) Int. Cl.5: G06F 15/401, H03M 7/42

(30) Priority: 09.12.88 GB 8828796

(43) Date of publication of application: 27.06.90 Bulletin 90/26

(84) Designated Contracting States: AT BE CH DE ES FR GB GR IT LI LU NL SE

(71) Applicant: BRITISH TELECOMMUNICATIONS public limited company
British Telecom Centre, 81 Newgate Street
London EC1A 7AJ(GB)

(72) Inventor: Clark, Alan Douglas
9 Meadowlands Kirton
Ipswich Suffolk IP10 0PP(GB)

(74) Representative: Semos, Robert Ernest Vickers et al
BRITISH TELECOM Intellectual Property Unit
13th Floor 151 Gower Street
London WC1E 6BA(GB)

(54) Data compression.

(57) In a data compression system including a dynamically compiled dictionary the dictionary is updated by storing a string comprising a word and a part-word. The part-word comprises a plurality of characters from the string immediately following the word in the input stream. The system may use a Mayne single-pass data compression algorithm. In a system using the Mayne algorithm after each string is matched the corresponding index is transmitted.

EP 0 375 221 A1

DATA COMPRESSION

The present invention relates to data compression systems which may be used, for example, to reduce the space required by data for storage in a mass storage device such as a hard disc, or to reduce the bandwidth required to transmit data. The invention is particularly concerned with data compression systems using dynamically compiled dictionaries. In such systems an input data stream is compared with strings stored in a dictionary. When characters from the data stream have been matched to a word in the dictionary the code for that word is read from the dictionary and transmitted in place of the original characters. At the same time when the input data stream is found to have character sequences not previously encountered and so not stored in the dictionary then the dictionary is updated by making a new entry and assigning a code to the newly encountered character sequence. This process is duplicated on the transmission and reception sides of the compression system. The dictionary entry is commonly made by storing a pointer to a previously encountered string together with the additional character of the newly encountered string.

According to a first aspect of the present invention in a data compression system including a dynamically compiled dictionary, the dictionary is updated by storing a string comprising a word and a part-word, the part-word comprising a plurality of characters from the string immediately following the word in the input data stream.

Preferably the part-word comprises only two characters.

Preferably the data compression system compresses data using a Mayne single-pass data compression algorithm.

In known systems dictionary entries are made either by combining the single unmatched character left over by the process of searching for the longest string match with the preceding matched word or by making entries comprising pairs of matched words. The former is exemplified by the Ziv Lempel algorithm, the latter by the conventional Mayne algorithm. The preferred example of the present invention uses the Mayne data compression algorithm but modifies the process of updating the dictionary so that the entries comprise one word and part of the following word rather than pairs of whole words. The inventor has found that this confers significant advantages in the efficiency of operation of the algorithm and that these advantages are particularly marked in the case where just two characters from the second of the pair of words are taken to form the dictionary entry.

According to a second aspect of the present invention, in a data compression system using the Mayne algorithm, during the updating of the dictionary after each string matching process the index for the corresponding dictionary entry is transmitted

One embodiment of the present invention is described in detail and contrasted with the prior art in the following technical description.

<u>Technical description of BTYZ - a modified Mayne compression algorithm</u>

1. Introduction

The Mayne algorithm (1975) predates the Ziv Lempel algorithm by several years, and has a number of features which were not built in to the Ziv Lempel implementations until the 1980's. The Mayne algorithm was originally proposed as a two pass adaptive compression scheme, but may be adapted to single pass operation. This paper discusses a modified version of the algorithm, however some implementation details are contained in a copending British application no. 8815978. The resource requirements in terms of memory and processing time, are similar to those achieved by the modified Ziv Lempel algorithm.

2. Data structure and encoding

As with the Ziv Lempel algorithm, the Mayne algorithm represents a sequence of input symbols by a codeword. This is accomplished using a *dictionary* of known strings, each entry in the dictionary having a corresponding index number of codeword. The encoder matches the longest string of input symbols with a dictionary entry, and transmits the index number of the dictionary entry. The decoder receives the index number, looks up the entry in its dictionary, and recovers the string.

The most complex part of this process is the string matching or parsing performed by the encoder, as this

necessitates searching through a potentially large dictionary. If the dictionary entries are structured as shown below however, this process is considerably simplified. The structure shown in the Figure is a tree representation of the series of strings beginning with "t"; the initial entry in the dictionary would have an index number equal to the ordinal value of "t".

To match the incoming string "the quick..", the initial character "t" is read and the corresponding entry immediately located (it is equal to the ordinal value of "t"). The next character "h" is read and a search initiated amongst the dependents of the first entry (only 3 in this example). When the character is matched, the next input character is read and the process repeated. In this manner, the string "the" is rapidly located and when the encoder attempts to locate the next character, " ", it is immediately apparent that the string "the" is not in the dictionary. The index value for the entry "the" is transmitted and the string matching process recommences with the character " ". This is based on principles which are well understood in the general field of sorting and searching algorithms (see ref 4).

The dictionary may be dynamically updated in a simple manner. When the situation described above occurs, i.e. string S has been matched, but string S+c has not, the additional character c may be added to the dictionary and linked to entry S. By this means, the dictionary above would now contain the string "the", and would achieve improved compression the next time the string is encountered.

The two pass form of the Mayne algorithm operates in the following way:-

(a) Dictionary construction

Find the longest string of input symbols that matches a dictionary entry, call this the *prefix* string. Repeat the process and call this second matched string the *suffix* string. Append the suffix string to the previs string, and add it to the dictionary. This process is repeated until the entire input data stream has been read. Each dictionary entry has an associated frequency count, which is incremented whenever it is used. When the encoder runs out of storage space it finds the least frequently used dictionary entry and re uses it for the new string.

(b) Encoding

The process of finding the longest string of input symbols that matches a dictionary entry is repeated, however when a match is found the index of the dictionary entry is transmitted. In the two pass scheme the dictionary is not modified during encoding.

To make the Mayne algorithm single pass during the dictionary update process, after each string matching process the index for the corresponding dictionary entry is transmitted.

With small dictionaries experience has shown that appending the complete string causes the dictionary to fill with long strings which may not suit the data characteristics well. With large dictionaries (say 4096+ entries) this is not likely to be the case. By appending the first two characters of the second string to the first, performance is improved considerably. The dictionary update process is modified to append two characters if the suffix string is 2 or more characters in length, or one character if the suffix string is of length 1.

```
match(entry, input stream, character)
    string - character
    entry - ordinal value of character
    do(
        read next character from input stream and append to string
        search dictionary for extended string
        if extended string is found
          then
            entry = index of matched dictionary entry)
    while ( found )
    /* returns with entry = last matched entry, character = last character
    read */
    return
```

```
encode( input stream )
    do(
        /* match first string */
        match( entry, input stream, character )
        output entry
        /* match second string */
        match(entry, input stream, character )
        output entry
        append initial two characters of second entry to first and
            add to dictionary
        )
    while( data to be encoded )
```

For example, if the dictionary contains the strings "mo", "us" and the word "mouse" is to be encoded using the single pass version of the Mayne algorithm.

(i) Read "m" and the following character "o" giving the extended string "mo".

(ii) Search in the dictionary for "mo" which is present, hence let *entry* be the index number of the string "mo".

(iii) Read the next character "u", which gives the extended string "mou".

(iv) Search the dictionary for "mou", which is not present.

(v) Transmit *entry* the index number of string "mo".

(vi) Reset the string to "u", the unmatched character.

(vii) Read the next character "s", giving the string "us".

(viii) Search the dictionary, and assign the number of the corresponding dictionary entry to *entry*.

(ix) Read the next character "e", giving the extended string "use".

(x) Search the dictionary for "use", which is not present.
(xi) Transmit *entry* the index number of string "us".
(xii) Add the string "mo" + "us" to the dictionary.
(xiii) Start again with the unmatched "e".
(xiv) Read the next character ......

Several other aspects of the algorithm need definition or modification before a suitable real time implementation is achieved. These relate to the means by which the dictionary is constructed, the associated algorithm, and storage recovery.

Many means for implementing the type of dictionary structure defined above are known. Knuth (1968) discusses some of these, others are of more recent origin. Two particular schemes will be outlined briefly:

(i) Trie structure

The above cited application on the modified Ziv Lempel algorithm discusses a trie structure [3] suitable for this application. This has been shown to provide a sufficiently fast method for application in modems. The scheme uses a linked list to represent the alternative characters for a given position in a string, and occupies approximately 7 bytes per dictionary entry.

(ii) Hashing

The use of hashing or scatter storage to speed up searching has been known for many years [4-6]. The principle is that a mathematical function is applied to the item to be located, in our case a string, which generates an address. Ideally, there would be a one-to-one correspondence between stored items and hashed adresses, in which case searching would simply consist of applying the hashing function and looking up the appropriate entry. In practice, the same address may be generated by several different data sets, collision, and hence some searching is involved in locating the desired items.

The key factor in the modified Mayne algorithm (and in fact in the modified Ziv Lempel algorithm) is that a specific searching technique does not need to be used. As long as the process for assigning new dictionary entries is well defined, an encoder using the trie technique can interwork with a decoder using hashing. The memory requirements are similar for both techniques.

Storage recovery is discussed in more detail in Section 4, in which a more efficient alternative to that described by Mayne is discussed.

3. Decoding

The decoder receives codewords from the encoder, recovers the string of characters represented by the codeword by using an equivalent tree structure to the encoder, and outputs them. It treats the decoded strings as alternately prefix and suffix strings, and updates its dictionary in the same way as the encoder.

It is interesting to note that the K*w* problem described in reference [7] does not occur in the Mayne algorithm. This problem occurs in the Ziv Lempel implementations of Welch, and Miller and Wegman, because the encoder is able to update its dictionary one step ahead of the decoder.

```
decode
        do(
            receive codeword
            look up entry in dictionary
            output string
            save string as prefix
            receive codeword
            look up entry in dictionary
            output string
            append first two characters of string to prefix and
                add to dictionary)
        while( data to be decoded )
```

4. Dictionary maintenance

4.a Dictionary updating

The encoder's dictionary is updated after each suffix string is encoded, and the decoder performs a similar function. New dictionary entries are assigned sequentially until the dictionary is full, thereafter they are recovered in a manner described below in section 4.b.

The dictionary contains an initial character set, and a small number of dedicated codewords for control applications, the remainder of the dictionary space being allocated for string storage. The first entry assigned is the first dictionary entry following the control codewords.

Each dictionary entry consists of a pointer and a character (see for example reference 3), and is linked to a parent entry in the general form shown in section 2. Creating a new entry consists of writing the character and appropriate link pointers into the memory locations allocated to the entry (see reference 6 for an example).

4.b Storage recovery

As the dictionary fills up it is necessary to recover some storage in order that the encoder may be continually adapting to changes in the data stream. The principal described in reference 6 has been applied to the modified Mayne algorithm, as it requires little processing overhead, and no additional storage.

When the dictionary is full entries are recovered by scanning the string storage area of the dictionary in simple sequential order. If an entry is a ***leaf***, i.e. is the last character in a string, it is deleted. The search for the next entry to be deleted will begin with the entry after the last one recovered. The storage recovery process is invoked after a new entry has been created, rather than before, this prevents inadvertent deletion of the matched entry.

5. Automatic Transparency

Not all data is compressible, and even compressible files can contain short periods of uncompressible data. It is desirable therefore that the data compression function can automatically detect loss of efficiency, and can revert to non-compressed or transparent operation. This should be done without affecting normal throughput if possible.

There are two modes of operation, *transparent* mode, and *compressed* mode.

(i) Transparent mode

(a) Encoder

The encoder accepts characters from the DTE interface, and passes them on in uncompressed form. The normal encoding process is however maintained, and the encoder dictionary updated, as described above. Thus the encoder dictionary can be adapting to changing data characteristics even when in transparent mode.

(b) Decoder

The decoder accepts uncompressed characters from the encoder, passes the characters through to the DTE interface, and performs the equivalent string matching function. Thus the decoder actually contains a copy of the encoder function.

(c) Transition from transparent mode

The encoder and decoder maintain a count of the number of characters processed, and the number of bits that these would have encoded in, if compression had been on. As both encoder and decoder perform the same operation of string matching, this is a simple process. After each dictionary character count is tested. When the count exceeds a threshold, *axf_delay*, the compression ratio is calculated. If the compression ratio is greater than 1, compression is turned ON and the encoder and decoder enter the *compressed* mode.

(ii) Compressed mode

(a) Encoder

The encoder employs the string matching process described above to compress the character stream read from the DTE interface, and sends the compressed data stream to the decoder.

(b) Decoder

The decoder employs the decoding process described in Section 3 to recover character strings from received codewords.

(c) Transition to transparent mode

The encoder arbitrarily tests its effectiveness, or the compressibility of the data stream, possibly using the test described above. When it appears that the effectiveness of the encoding process is impaired the encoder transmits an explicit codeword to the decoder to indicate a transition to compressed mode. Data from that point on is sent in transparent form, until the test described in (1) indicates that the system should revert to compressed mode.
The encoder and decoder revert to *prefix* mode after switching to transparent mode.

6. Operation on DTE timeout, end of frame, or end of message

A *flush* operation is provided to ensure that any data remaining in the encoder is transmitted. this is needed as there is a bit oriented element to the encoding and decoding process which is able to store fragments of one byte. The next data to be transmitted will therefore start on a byte boundary. When this

operation is used, which can only be in compressed mode, an explicit codeword is sent to permit the decoder to realign its bit oriented process. This is used in the following way:-

(a) DTE timeout

When a DTE timeout or some similar condition occurs, it is necessary to terminate any string matching process and flush the encoder. The steps involved are .. exit from string matching process, send codeword corresponding to partially matched string, send FLUSHED codeword and flush buffer.

(b) End of Buffer

At the end of a buffer the flush process is not used, unless there is no more data to be sent. The effect of this is to allow codewords to cross frame boundaries. If there is no more data to be sent, the action defined in (a) is taken.

(c) End of message

The process described in (a) is carried out.

7. Examples

(i) Dictionary update

Input string "elephant", strings already in dictionary "ele", "ph"

Match "ele" and "ph"
Add "p" to "ele" and "h" to "elep", giving "eleph"

Match "a" and "n"
Add "n" to "a" giving "an"

Match "t" and " "
Add " " to "t" giving "t "

(ii) Transparency check
input string "elephant" with matching process as above

Match "ele", increase character count by 3
Increase compressed bit count by length of codeword
Match "ph", increase character count by 2
Increase compressed bit count by length of codeword
Add new dictionary entry
If character count > test interval
check compressed bits against character count • octet size (8)
if in transparent mode and compressed bits < ch count • 8
switch to compressed mode
else
if in compressed mode and compressed bits > ch count • 8
switch to transparent mode
reset counts

i.e. "ele" + "ph", "a" + "n", "t" + " "
^ test here and ^ to see if character count exceed test interval, if it does then compare performance.

(iii) Flush operation
Input string as above, however timeout occurs after "p"

(a) Compressed mode
Send codeword for currently matched string, even if incomplete .. entry corresponding to "p"
Send FLUSHED codeword

```
Flush bits from bitpacking routing, and re-octet align
Reset encoder to prefix state
Reset string to null state, and wait for next character
```

```
(b) Transparent
Send existing buffer contents
```

8. Implementation aspects

The algorithm is comparable in complexity to the modified Ziv Lempel algorithm. The memory requirement is 2 bytes per dictionary entry for the first 260 dictionary entries, and 7 bytes per entry thereafter giving, for each of the encoder and decoder dictionaries:-

1024 entries: 6k bytes
2048 entries: 13k bytes
4096 entries: 27 kbytes

Processing speed is very fast. For operation at speeds up to 38.4 kbits over a 9.6 kbit link only a Z80 family processor should be required.

Response time is minimized through the use of a *timeout* codeword, which permits the encoder to detect intermittent traffic (i.e. keyboard operation) and transmit a partially matched string. This mechanism does not interfere with operation under conditions of continuous data flow, when compression efficiency is maximized.

9. Summary

The algorithm described above is ideally suited to the modem environment, as it provides a high degree of compression but may be implemented on a simple inexpensive microprocessor with a small amount of memory.

A range of implementations are possible, alowing flexibility to the manufacturer in terms of speed, performance and cost. This realises the desire of some manufacturers to minimise implementation cost, and of others to provide top performance. The algorithm is however well defined and it is thus possible to ensure compatibility between different implementations.

References

[1] Information Compression by Factorizing Common Strings
A. Mayne, E.B. James
Computer Journal, Vol 18.2 pp 157 - 160, 1975
[2] Compression of Individual Sequences via Variable Rate Coding
J. Ziv, A. Lempel
IEEE Trans, IT 24.5, pp 530 = 536, 1978
[3] Use of Tree Structures for Processing Files
E.H. Sussenguth
CACM, Vol 6.5 pp 272 - 279, 1963
[4] The Art of Computer Programming, Vol 3 Sorting and Searching
D. Knuth, Addison Wesley, 1968
[5] Scatter Storage Techniques
R. Morris
CACM, Vol 11.1 pp 38 - 44, 1968
[6] Technical Description of BTLZ

British Telecom contribution to CCITT SGXVII
[7] A technique for High Performance Data Compression
T Welch
Computer, June 1984, pp 8 - 19

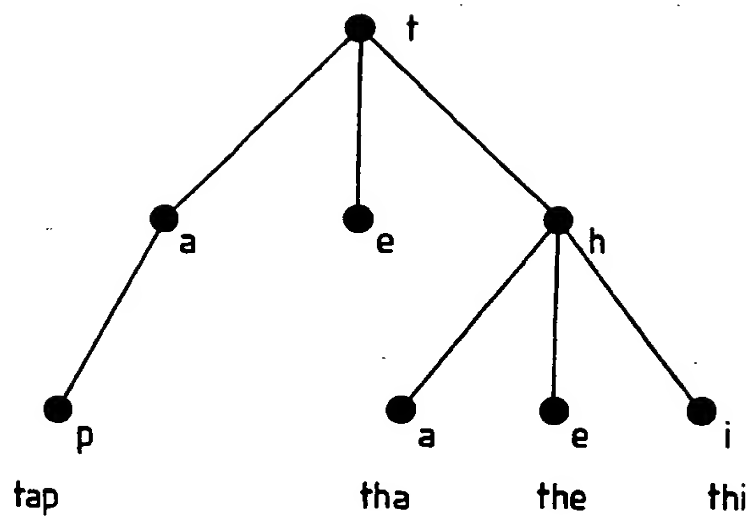
Claims

1. A data compression system including a dynamically compiled dictionary, characterised in that the dictionary is updated by storing a string comprising a word and a part-word, the part-word comprising a plurality of characters from the string immediately following the word in the input data stream.

2. A system according to claim 1, in which the part-word comprises only two characters.

3. A system according to claim 1 or 2, in which the data compression system compresses data using a Mayne single-pass data compression algorithm.

4. A data compression system using the Mayne algorithm, in which during the updating of the dictionary after each string matching process the index for the corresponding dictionary entry is transmitted.


t
a
e
h
p
a
e
i
tap
tha
the
thi

European Patent Office

EUROPEAN SEARCH REPORT

Application Number

EP 89 31 2818

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int. Cl.5)
X	EP-A-0 127 815 (IBM) * Page 11, line 13 - page 12, line 14; claim 1 *	1,3,4	G 06 F 15/401 H 03 M 7/42
Y		2	
Y	EP-A-0 280 549 (OKI ELECTRIC INDUSTRY CO.) * Claims 1-4 *	2	
D,A	COMPUTER JOURNAL, vol. 18, no. 2, 1975, pages 157-160; A. MAYNE et al.: "Information compression by factorising common strings" * The whole document *	1-4	
A	PROC. VERY LARGE DATABASES, Cannes, 9th - 11th September 1981, pages 435-447, IEEE, New York, US; C.A. LYNCH et al.: "Application of data compression techniques to a large bibliographic database" * Page 441, left-hand column, lines 58-65 *	2	

TECHNICAL FIELDS SEARCHED (Int. Cl.5)

G 06 F 15/00
H 03 M 7/00

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	22-03-1990	POPINEAU G.J.P.

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

THIS PAGE BLANK (USPTO)